Getting Started with R

Alessio Palmisano

1. Introduction

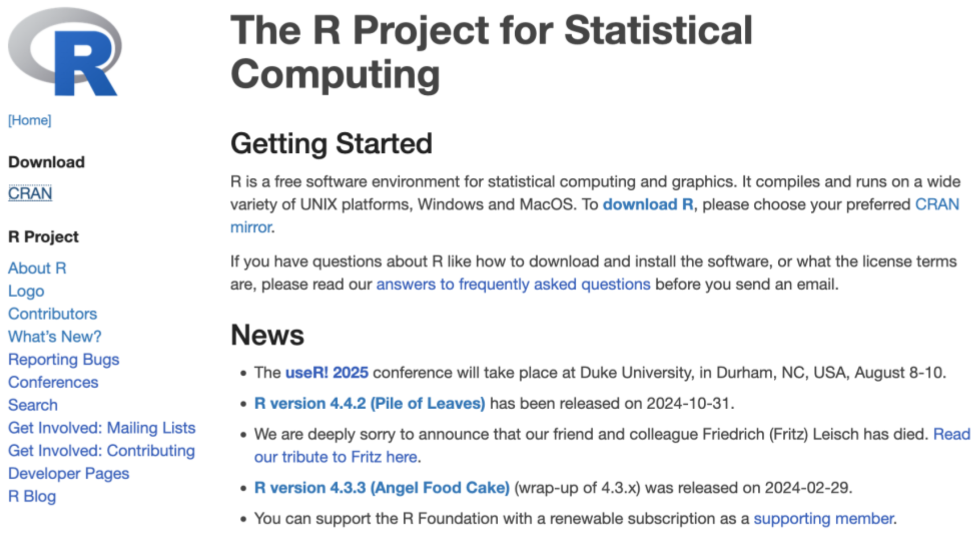
This tutorial serves as a guide to programming in R, taking you from data loading to creating your own functions that can outperform those of many other R users. R is a free, open-source programming language designed for data management and analysis. It offers extensive tools for statistical and graphical applications, with its capabilities continually growing. Beyond the core installation, R users from around the globe contribute numerous packages that extend its functionality. The software is freely accessible and available in compiled versions for Windows, Mac OS X, and various Linux distributions, including Debian, Redhat, Suse, and Ubuntu. This tutorial will introduce you to installing R, executing commands, and finding help.

1.1 Installing R

Before starting, you'll need to learn a couple of things: RStudio serves as a tool for interacting with your computer, while R provides the language to convey your instructions. Of course, you may just install just R and skip Rstudio installation. However, in this course we will install and use both.

To begin, you'll need to install a recent version of R (version 4.2 or later is recommended). Follow these steps based on your operating system:

- Visit the R Project website and click on the CRAN link under the "Downloads" section on the left side of the page (Figure 1).
- Select a download mirror, or one in your country for faster access.
- Choose your operating system (Windows, macOS, or your specific Linux distribution) and follow the corresponding instructions provided.



The R Project for Statistical Computing

Getting Started

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To **download R**, please choose your preferred CRAN mirror.

If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

News

- The **useR! 2025** conference will take place at Duke University, in Durham, NC, USA, August 8-10.
- **R version 4.4.2 (Pile of Leaves)** has been released on 2024-10-31.
- We are deeply sorry to announce that our friend and colleague Friedrich (Fritz) Leisch has died. Read our tribute to Fritz here.
- **R version 4.3.3 (Angel Food Cake)** (wrap-up of 4.3.x) was released on 2024-02-29.
- You can support the R Foundation with a renewable subscription as a supporting member.

Figure 1. The R Project website

1.2 Installing RStudio

RStudio is a powerful integrated development environment (IDE) designed for R, Python, and other programming tools. It offers enhanced features for coding, debugging, and managing data, making it an essential tool for handling large and complex R projects.

To install RStudio, follow these steps:

*Visit the official RStudio website and click on the "Download" option at the top of the page.

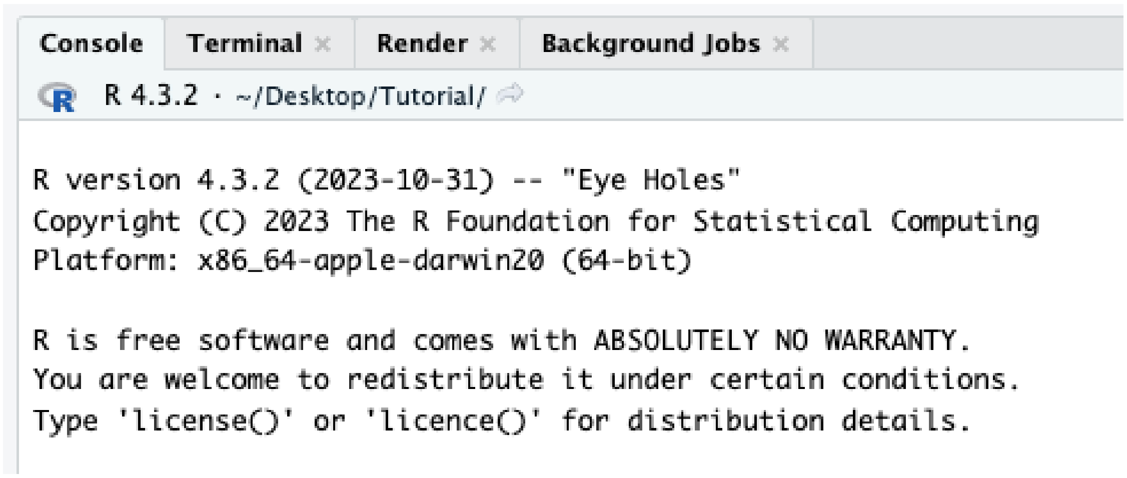
*Choose the RStudio Desktop version.

*Download the latest installer file that matches your operating system.

*Run the installer and follow the prompts to complete the setup. RStudio will typically detect your existing R installation automatically.

1.3 Launch RStudio

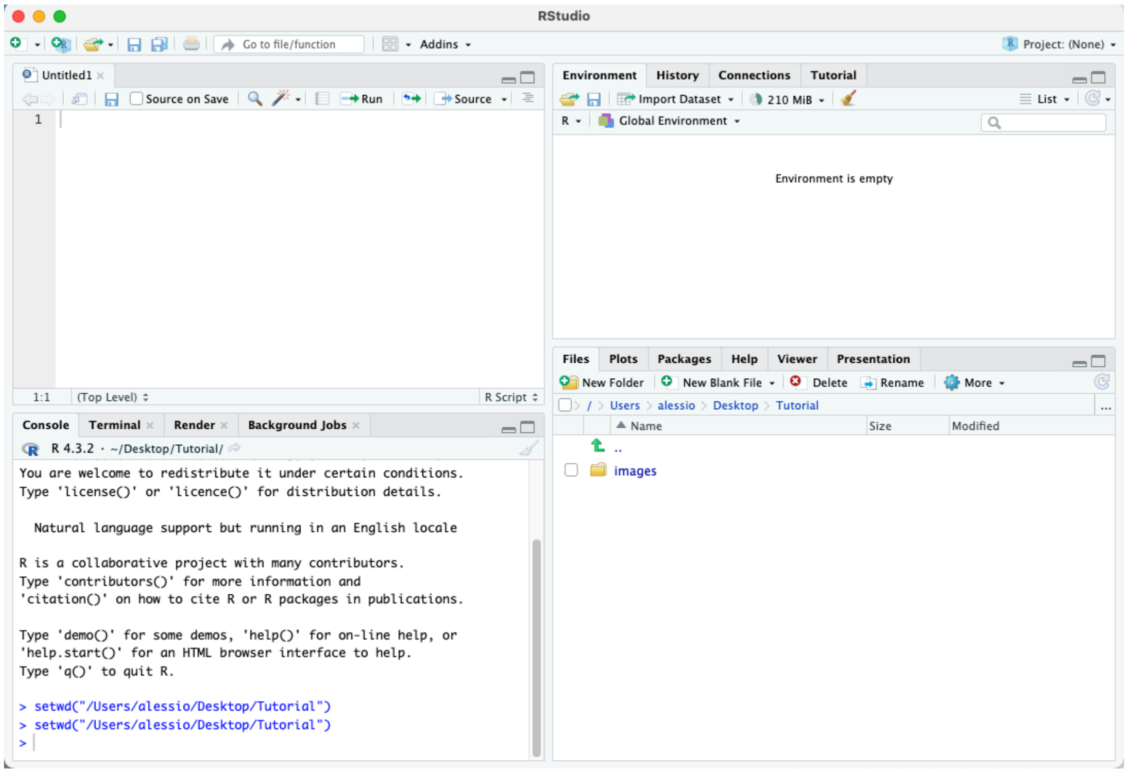
After installing both R and RStudio, launch RStudio and locate the Console window, usually found on the left side of the screen. The Console will display the version of R linked to your RStudio installation. Ideally, this should match the recent version of R you just installed (see Figures 2)



```
R version 4.3.2 (2023-10-31) -- "Eye Holes"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
```

Figure 2. The console panel and the R version installed

Let’s start by exploring the R-Studio setup. Upon launching R-Studio for the first time, you’ll notice the interface is divided into three sections. Before you begin, click on “File” at the top, then select “New File > R Script” to open a fourth section. The screen should resemble the one shown below (Figure 3).



```
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> setwd("/Users/alessio/Desktop/Tutorial")
> setwd("/Users/alessio/Desktop/Tutorial")
>
```

Figure 3. The RStudio IDE for R

Here’s a breakdown of the different panes in R-Studio:

- Workspace: Located in the top-left pane, this section allows you to write your code and other documents before executing them.

- Console: Found in the bottom-left pane, the console lets you type and run commands directly. Any code executed from the workspace will also appear here.
- Environment/History: The upper-right pane contains two tabs: Environment, which lists the objects and functions currently initialized, and History, which shows a record of commands previously run in the console.
- Files/Plots/Packages/Help: The lower-right pane has several tabs, including Files (which displays files in the current directory), Plots (where visualizations created in the console will appear), Packages (a list of installed and initialized R packages), and Help (providing details on specific functions and packages).

If you prefer a different layout or want to adjust the visibility of these panes, you can do so by navigating to "View > Panes" and selecting the options you prefer.

2. First steps with R

2.1 Mathematical operations

The RStudio interface is straightforward to use. To execute R code, simply type it into the bottom line of the console pane and press Enter. The code you write is referred to as a command because it instructs your computer to perform a specific action. This line is known as the command line. When you enter a command at the prompt and hit Enter, the computer processes it and displays the results. Try entering the following mathematical operations in the console:

```
1+1

## [1] 2

10/2

## [1] 5

3*10

## [1] 30

50-30

## [1] 20

9^2

## [1] 81
```

In R, parentheses ( ) are used to group operations and define the order in which calculations are performed. They can also be nested to handle more complex expressions. Type the equations below

```
((10+5)*3)/5

## [1] 9

((10*9)-(2*3))/((1+3)*2)

## [1] 10.5

((10*9)-(2*3))/(1+3)*2

## [1] 42
```

Certain commands in R generate multiple values as output, with results spanning several lines. The colon operator (:) generates a sequence of integers between two specified numbers, including both endpoints. It's a simple and efficient way to create a range of numbers. For instance, using the command 1:20 produces a sequence of 20 integers, starting at 1 and ending at 30.

```
1:20

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

If you enter an incomplete command and press Enter, R will display a + prompt. This indicates that R is waiting for you to complete the command. You can either finish typing the command or press the Escape key to cancel and start fresh.

10 + +

When you enter a command that R does not understand, it will display an error message. There's no need to worry if this happens—R is simply indicating that it couldn't process your request. You can try a different command at the next prompt.

5 % 10 Error: unexpected input in "5 % 10"

IMPORTANT: in R, the hashtag symbol (#) has a unique function—it tells R to ignore anything written after it on the same line. This makes it a valuable tool for adding comments or notes to your code. While these comments are readable by humans, R will skip over them during execution. In R, the hashtag is commonly referred to as the commenting symbol.

2.2 Logical operators

R provides logical operators (as listed below) that return values of TRUE or FALSE. These operators can be combined with other operations and are particularly useful in constructing more advanced functions and conditional statements, as demonstrated later.

Logical Operators in R

Operator	Meaning
<	less than
>	greater than

Operator	Meaning
<=	less than or equal to
>=	greater than or equal to
==	exactly equal to
!=	not equal to
&	and
|	or

Table 1. Table showing the main logical operators.

```
50 < 20

## [1] FALSE

50 > 20

## [1] TRUE

50 + 20 == 70

## [1] TRUE

50 & 20 > 30

## [1] FALSE

50 | 20 > 30

## [1] TRUE
```

3. Objects

In R, you can store data by assigning it to an object. But what exactly is an object? It's essentially a label that allows you to access stored data later. For instance, you could assign data to objects such as "a" or "b". Whenever R encounters one of these objects, it substitutes it with the data it contains. To create an object in R, simply pick a name and use the combination of a less-than sign < followed by a hyphen -, which resembles an arrow (<-), to assign data to it. R will create the object with the specified name and store the data that comes after the arrow within it. Follow the example below

```
a <-5
a

## [1] 5
```

Since the object "a" was previously assigned the value of 5, you are now adding 2 to the object named "a".

```
a + 2
```

```
## [1] 7

b <- 1:10
b

##  [1]  1  2  3  4  5  6  7  8  9 10

a + b

##  [1]  6  7  8  9 10 11 12 13 14 15
```

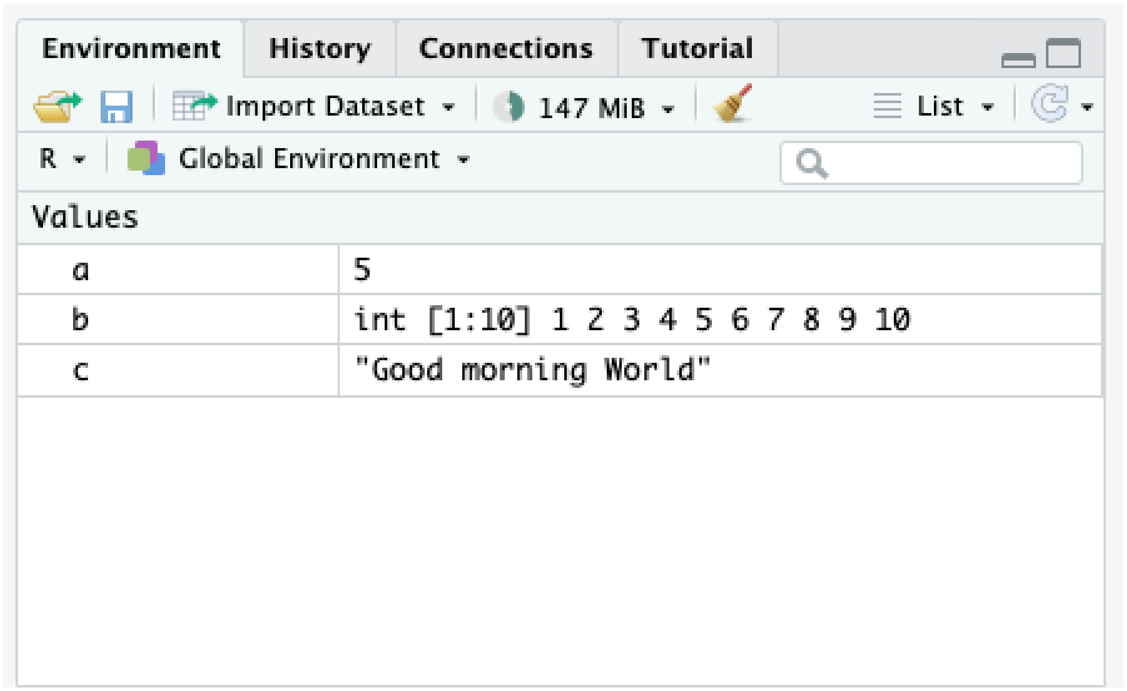
You can also create objects containing characters rather than numbers.

```
c <- "Good morning World"
c

## [1] "Good morning World"
```

You can check the object names you have already created by using the `ls` function:

```
ls()

## [1] "a" "b" "c"
```

If you are using RStudo, new objects gets stored under the tab "Environment" on the upper-right pane. This pane displays all the objects you have created since launching RStudio (see Figure 4).



Values

a	5
b	int [1:10] 1 2 3 4 5 6 7 8 9 10
c	"Good morning World"

In R, object names are case-sensitive and cannot contain spaces. They can include both letters and numbers but **must begin with a letter**. Using clear and descriptive object names is recommended, especially for objects that will be used repeatedly.

When it comes to naming conventions, there are several commonly used styles:

- good_morning_world
- GoodMorningWorld
- good-morning-world

Furthermore, it's advisable to avoid using periods (.) or special simbols (e.g.,^, !, $, @, +, -, /, *) to separate words in object names, as they have specific uses in R functions and can create unnecessary confusion.

IMPORTANT: Avoid overwriting an R object by assigning an object the same name.

The four most common object types in R are vectors, matrices, lists, and data frames.

- **Vectors**: A collection of values, all of the same type (e.g., character, numeric, etc.). If numeric and character data are mixed, R will interpret all entries as character data.
- **Matrices**: A rectangular, two-dimensional table of values, all of the same type (e.g., character, numeric, etc.).
- **Data Frames**: A rectangular, two-dimensional table where each column can hold a different data type (e.g., one column may be numeric while another is character).
- **Lists:** A collection of R objects (which can include vectors, matrices, data frames, or others) combined into a single object. Lists can store elements of various formats and types.

You can remove objects using the `rm()` function. You follow the syntax: `rm(name of the object to delete)`

3.1 Vectors

A vector is a basic data structure that represents a collection of elements. All the elements in a vector must be of the same type, such as numbers, characters, or logical values. Vectors are one of the most common data structures in R and are used extensively in data analysis, manipulation, and computation.

R has different classes of data, but here we provide the most common categories:

- **numeric**: integers, floating-point numbers.
- **character**: words or sentences.
- **logical**: TRUE or FALSE values.
- **factors**: data type used to represent categorical variables, which have a fixed number of unique values called levels. Factors are useful when dealing with

variables that represent categories or groups, like gender, color, or yes/no responses.

You can assign a vector of numbers or characters to an object by using the function `c()`, which stands for combine. See some examples below:

```
luke<-c(1,2,3,4,5)
luke

## [1] 1 2 3 4 5

sky<-c(1:5,10:15,50:55)
sky

##  [1]  1  2  3  4  5 10 11 12 13 14 15 50 51 52 53 54 55

walker<-c("red","black","white","blue")
walker

## [1] "red"   "black" "white" "blue"
```

You can also create a new object by manipulating a former one as follows:

```
obj4<-luke * 3
obj4

## [1]  3  6  9 12 15
```

To access a specific value or a set of values in a vector, you can use square brackets [ ] and specify the item(s) you want to retrieve.

```
sky[10] # it would return the tenth element from object sky

## [1] 14

sky[10:15] # it would return items 10 trough 15 in object sky

## [1] 14 15 50 51 52 53

sky[c(15,9,1)] # items 15, 9, 1 in that order from object sky

## [1] 53 13  1
```

It is also possible to search through vectors or other objects to find particular values.

```
walker[walker == "black"]

## [1] "black"
```

To check if a specific value exists within an object, you can use the `%in%` operator, which will return a logical value.

```
"red" %in% walker
```

```
## [1] TRUE
```

```
"yellow" %in% walker
```

```
## [1] FALSE
```

Vectors can be used in various operations, including arithmetic and logical operations, element-wise. For example:

```
c(1,2,3) * c(4,5,6)
```

```
## [1]  4 10 18
```

```
c(TRUE, FALSE) & c(FALSE, TRUE)
```

```
## [1] FALSE FALSE
```

```
c(TRUE, FALSE) & c(TRUE, FALSE)
```

```
## [1]  TRUE FALSE
```

You can determine the number of elements in a OBJECT using the `length()` function:

```
length(sky)
```

```
## [1] 17
```

There are cases when different classes of R vectors are combined, either unintentionally or deliberately. Let's explore what happens when the following code is executed.

```
test<-c(1,10,"a")
test
```

```
## [1] "1"  "10" "a"
```

When different items are combined in a vector, R automatically applies coercion to ensure that all elements in the vector belong to the same class. In the example above, implicit coercion takes place. R tries to find a way to represent all items in the vector in a compatible manner. This process can work as expected at times, but not always. For instance, if you combine a numeric element with a character element, R will convert the entire vector into a character vector, as numbers can generally be represented as strings.

We can check the class of the object created by using the command `class()`.

```
class(test)
```

```
## [1] "character"
```

Objects can be explicitly converted from one class to another using the `as.*` functions, when those functions are available for the specific class types.

```
as.numeric(test)
```

```
## Warning: NAs introduced by coercion
## [1]  1 10 NA
```

In cases where R is unable to determine how to coerce an object, it may result in the creation of NA values. This is why it is important to avoid mixing different classes of elements within the same vector.

3.2 Matrices

R is also well-suited for handling tabular data. While large datasets are often read from external files (as discussed in the "working with files" section), simple numeric tabular data can be created directly in R using the `matrix()` function. For instance, you can create a two-row, two-column matrix by transforming a numeric vector into a matrix and specifying the number of rows (nrow) and columns (ncol). The values you assign within the `matrix()` function are known as arguments.

```
neo<-matrix(data = c(5,15,28,40), nrow = 2, ncol = 2)
neo
```

```
##      [,1] [,2]
## [1,]    5   28
## [2,]   15   40
```

Note that by default, the `matrix()` function populates values column by column, meaning it fills the first column first, then proceeds to the next. To modify this setting, you can use the argument "byrow" which is set to FALSE by default. Let's set that option to TRUE and observe the results.

```
neo<-matrix(data = c(5,15,28,40), nrow = 2, ncol = 2, byrow = TRUE)
neo
```

```
##      [,1] [,2]
## [1,]    5   15
## [2,]   28   40
```

If you want to name the columns and the names of the matrix in R, you can use the `rownames()` function for rows and `colnames()` for columns. Here is an updated example.

```
colnames(neo)<-c("Width", "Lenghth")
rownames(neo)<-c("artefact1", "artefact2")
neo
```

```
##           Width Lenghth
## artefact1     5      15
## artefact2    28      40
```

Matrices can also be utilized with various mathematical and statistical functions that are built directly into R. For instance, let's use the function `summary()` to generate a summary of key statistics or information about the data stored in our matrix.

```
summary(neo)
```

```
##      Width           Lenghth     
##  Min.   : 5.00   Min.   :15.00  
##  1st Qu.:10.75   1st Qu.:21.25  
##  Median :16.50   Median :27.50  
##  Mean   :16.50   Mean   :27.50  
##  3rd Qu.:22.25   3rd Qu.:33.75  
##  Max.   :28.00   Max.   :40.00
```

If you want to know the size of a matrix, you can use the function `dim()`.

```
dim (neo)

## [1] 2 2
```

3.3 Data frames

Like a table or spreadsheet, a data frame has rows and columns. Each row typically represents an observation (or data point), and each column represents a variable or feature. Unlike matrices, where all elements must be of the same type, columns in a data frame can contain different types of data. For example, one column could be numeric (e.g., width), another could be a factor (e.g., material), and another could be a character vector (e.g., names).

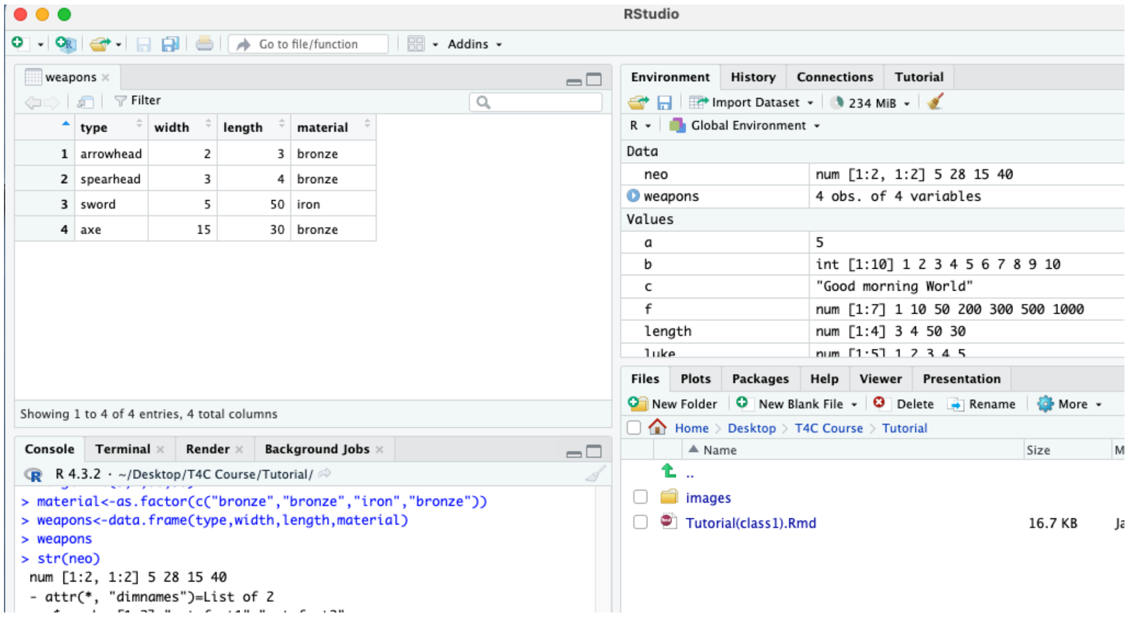
A data frame can be created by combining a set of vectors. For example:

```
type<-c("arrowhead","spearhead", "sword", "axe")
width<-c(2,3,5,15)
length<-c(3,4,50,30)
material<-as.factor(c("bronze","bronze","iron","bronze"))
weapons<-data.frame(type,width,length,material)
weapons

##        type width length material
## 1 arrowhead     2      3   bronze
## 2 spearhead     3      4   bronze
## 3     sword     5     50     iron
## 4       axe    15     30   bronze
```

To examine the data type of each column in a data frame, we can use the `str()` function or the `structure()` function in R.

```
str(weapons)

## 'data.frame':    4 obs. of  4 variables:
##  $ type    : chr  "arrowhead" "spearhead" "sword" "axe"
##  $ width   : num  2 3 5 15
##  $ length  : num  3 4 50 30
##  $ material: Factor w/ 2 levels "bronze","iron": 1 1 2 1
```

Once you've created a data frame in RStudio, you can view it by clicking on its name under the *Environment* tab in the upper-right pane. The data frame will then appear in the upper-left pane for easy visualization (figure 5).



	type	width	length	material
1	arrowhead	2	3	bronze
2	spearhead	3	4	bronze
3	sword	5	50	iron
4	axe	15	30	bronze

Showing 1 to 4 of 4 entries, 4 total columns

```
Data
neo       num [1:2, 1:2] 5 28 15 40
weapons   4 obs. of 4 variables
Values
a         5
b         int [1:10] 1 2 3 4 5 6 7 8 9 10
c         "Good morning World"
f         num [1:7] 1 10 50 200 300 500 1000
length    num [1:4] 3 4 50 30
luke      num [1:5] 1 2 3 4 5
```

```
Home > Desktop > T4C Course > Tutorial
Name                    Size
..
images
Tutorial(class1).Rmd    16.7 KB
```

```
R 4.3.2 · ~/Desktop/T4C Course/Tutorial/
> material<-as.factor(c("bronze","bronze","iron","bronze"))
> weapons<-data.frame(type,width,length,material)
> weapons
> str(neo)
 num [1:2, 1:2] 5 28 15 40
 - attr(*, "dimnames")=List of 2
```

3.4

Lists

A list is a versatile data structure that can hold a collection of elements of different types. Put simply, it combines multiple objects into a single one. Unlike vectors or data frames, which store elements of the same type (e.g., all numbers, all characters), a list can contain a mix of different data types, such as numbers, characters, vectors, data frames, or even other lists.

Each element in a list is called a "component," and they are indexed by position or by name (if assigned). Lists are especially useful when you want to organize different types of data that are related but don't fit neatly into a single data structure like a vector or matrix.

Lists can be defined using the `list()` function. For example let us create a list of the objects we have created above.

```
list1<-list(a,b,c,neo,weapons)
list1

## [[1]]
## [1] 5
##
## [[2]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[3]]
## [1] "Good morning World"
##
## [[4]]
##             Width Lenghth
## artefact1     5      15
## artefact2    28      40
##
## [[5]]
##        type width length material
## 1 arrowhead     2      3   bronze
## 2 spearhead     3      4   bronze
## 3     sword     5     50     iron
## 4       axe    15     30   bronze
```

To access a specific element within a list, you use double square brackets “[[]]” and provide the numeric index of the element you want to retrieve.

```
list1[[2]]

##  [1]  1  2  3  4  5  6  7  8  9 10

list1[[5]]

##        type width length material
## 1 arrowhead     2      3   bronze
## 2 spearhead     3      4   bronze
## 3     sword     5     50     iron
## 4       axe    15     30   bronze
```

You can also combine double and single square brackets to access specific items within elements of a list. For example:

```
list1[[2]] [7] #item 7 in list object number 3

## [1] 7

list1[[5]] [4,1] #row 4 column 1 in list object number 5

## [1] "axe"

list1[[5]] [4,] # row 4 and all columns in list object number 5

##   type width length material
## 4  axe    15     30   bronze

list1[[5]] [,1] # all rows and column 1 in list object number 5

## [1] "arrowhead" "spearhead" "sword"     "axe"
```

4. Functions

4.1 Using basic functions

R provides a variety of built-in functions designed to handle common operations and perform statistical analyses. Using a function is straightforward: simply write the function's name followed by the data it should process, enclosed in parentheses. Here a list of examples:

```
f<-c(1,10,50,200,300,500,1000)
f

## [1]    1   10   50  200  300  500 1000

min(f)

## [1] 1

max(f)

## [1] 1000

mean(f)

## [1] 294.4286

median(f)

## [1] 200

round(mean(f))

## [1] 294

round(mean(f),digits = 2)

## [1] 294.43
```

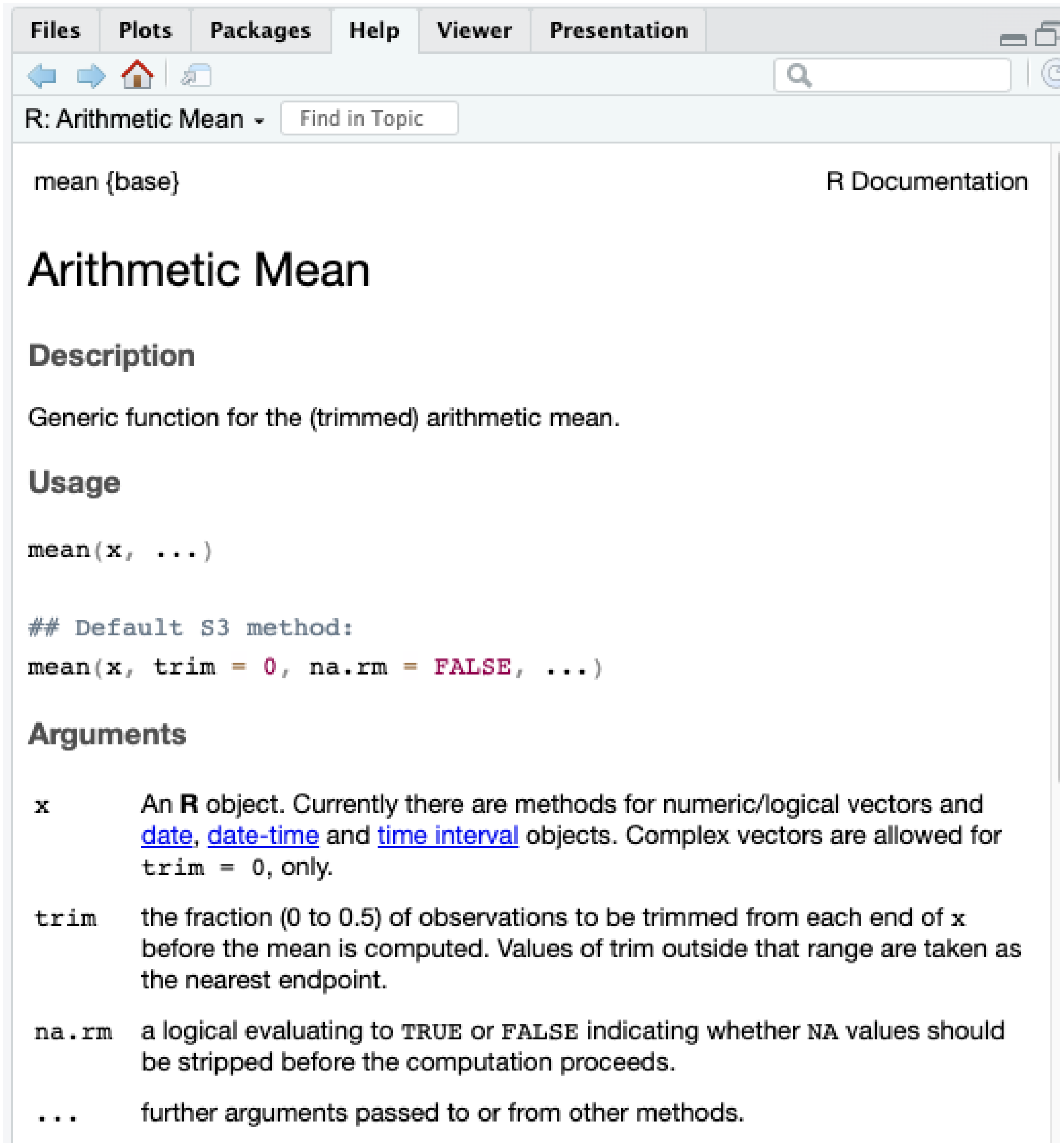
R includes more than 1,000 core functions, with new ones being added regularly. This makes it challenging to remember and master all of them. Fortunately, every R function has a dedicated help page that you can easily access. To view it, simply type help("name of function") or ?name of function in the console. For example

```
?mean
```

or

```
help("mean")
```

In the lower-left pane, under the "Help" tab, you can find detailed information about the function you requested (Figure 6).



mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An **R** object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical evaluating to `TRUE` or `FALSE` indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Help pages in R are typically divided into sections, which may vary but often include:

- **Description**: A brief description of the function's purpose.
- **Usage**: An example of how to write the function, showing the expected order of arguments.
- **Arguments**: A list of the function's arguments, the required input types, and how they are used.
- **Details**: A deeper explanation of the function and important notes for its usage.
- **Value**: Information about what the function returns.
- **See Also**: Related functions for further exploration.

- **Examples**: Working example code that demonstrates the function in action. This is one of the most useful sections of a help page because these examples demonstrate how the function works and offer an excellent way to learn through practice.

If you want to find the help page for a function but can't remember its name, you can perform a keyword search by typing help.search("name of the function") or two question marks followed by the keyword in R's command line ??name of function. For example:

```
??mean
```

R will return a list of relevant functions related to the keyword. In addition, R has a highly active and supportive community of users that you can reach out to for help and resources.An excellent resource for R-related questions is Stack Overflow, a platform where programmers can answer queries, and users can rank answers based on their usefulness. You can either submit your own questions or search through the extensive archive of previously answered R-related queries.

4.2 writing your own functions

In this section, we will explore how to write your own function in R. When you find yourself repeatedly performing the same task, it's inefficient to copy and paste the code each time. Instead, it's more practical to define a function once and simply call it whenever needed. After being defined, a custom function operates just like the built-in functions or those from packages that we've previously discussed. Below is the fundamental syntax for creating a function in R:

```
function_name <- function(arguments) {
  result <- expression_to_evaluate #code to execute
  return(result)
}
```

Each function in R consists of three main components: a name, a set of arguments, and a body of code. To create your own function, you need to define these parts and store them in an R object, which can be done using the command `function()`.

Once the function is defined, you can run it as follows: function_name(arguments)

Let us try an example of a function summing two numbers.

```
sum_numbers<-function(x,y){
  sum<- x+y
  return(sum)
}
```

Just a brief explanatio of the code above:

- **sum_numbers** is the name of the function.
- **x** and **y** are the parameters (inputs) to the function.
- **sum <- x+y** is the calculation that happens inside the function.
- **return(sum)** ensures that the sum is returned when the function is called.

Once you defined the function, you can sum two numbers:

```r
sum_numbers(3,7)

## [1] 10
```

We can also create a function returning multiple values. In this example, we create a function providing some statistics from a set of values.

```r
calculate_stats <- function(numbers) {
  mean_value <- mean(numbers)
  sum_value <- sum(numbers)
  min_value <- min(numbers)
  max_value <- max(numbers)

  # Returning multiple values in a list
  result <- list(mean = mean_value, sum = sum_value, min = min_value, max =
max_value)
  return(result)
}

# Call the function
calculate_stats(c(1, 5, 10, 30, 100))

## $mean
## [1] 29.2
##
## $sum
## [1] 146
##
## $min
## [1] 1
##
## $max
## [1] 100
```

5. Advanced tools: control structures in R.

Control structures in R help determine how your code runs, allowing it to make decisions and repeat actions based on certain conditions. Instead of running the same code every time, control structures let your program react to different inputs or data.

Here are some common control structures in R:

- **if** and **else**: Check a condition and decide what to do based on the result.
- **for**: Repeats a task (loop) a set number of times.
- **while**: Keeps running a loop as long as a condition remains true.

These tools make your R code more flexible and efficient, allowing it to handle different situations automatically.

5.1 Conditional statements: "if" and "else"

The **if-else** combination is one of the most widely used control structures in R—and in programming in general. It helps you check a condition and decide what action to take based on whether the condition is **true** or **false**.

To begin with, you can use a simple if statement on its own.

Example 1:

```
x<-10
if (x > 5) {
  print("x is greater than 5")
}

## [1] "x is greater than 5"
```

In the example above, if the condition "x > 5" is evaluated as TRUE, the print command is executed by displaying the specified message. However, if the condition is FALSE, nothing happens.

Let us try Example 2:

```
x<-3
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is smaller than 5")
 }

## [1] "x is smaller than 5"
```

n Example 2, the if-else combination is used. If the first condition is evaluated as TRUE, the first print command is executed. However, if the condition is FALSE, then the second print command is executed by displaying the corresponding message. This ensures that one of the two possible actions always takes place, depending on the result of the statement.

The "if" and "else" conditions can be combined into the `ifelse()` function, which is a **vectorised conditional function**, which evaluates a condition for each element of a vector and return different values based on wheter the condition is TRUE or FALSE. It is particularly useful when you want to apply a condition to multiple elements at once.

For instance:

```
scores <- c(18, 25, 15, 16, 30)
grades <- ifelse(scores >= 18, "Pass", "Fail")
print(grades)

## [1] "Pass" "Pass" "Fail" "Fail" "Pass"
```

In the above example the grades of an university exam have been evaluated:

- 18, 25, and 30 are greater than or equal to 18 = "Pass"
- 15 and 16 are less than 18 = "Fail"

5.2 Loops

In R, loops are used to execute a block of code multiple times. The most common loop functions are **for** and **while**. A **for** loop is used when the number of iterations is known beforehand. The typical syntax is the following:

```
for (variable in sequence) {
  # Code to be executed
}
```

This means that for each value within a given sequence, the expression in the event chunk should be evaluated. To better understand this, let's go through a detailed example.

```
for (i in 1:5) { #for every value in the sequence from 1 to 5
  print(i+10)
}
```

```
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
```

This example demonstrates how the `for (i in 1:5)` statement works. It first assigns i = 1 and executes `print(i + 10)`, then moves to i = 2 and evaluates `print(i + 10)`, continuing this process until i = 5. The main characteristic of for loops is that they allow the iterator i to be used within the curly brackets {} to execute a statement across a specified range of values.

Instead, a **while** loop runs as long as a specified condition remains `TRUE` and it follows the following syntax.

```
while (condition) {
  # Code to be executed
}
```

Let us run an example:

```
i <- 1  # Initialize i

while (i <= 5) {  # Condition to run the loop
  print(i + 10)  # Perform the operation
  i <- i + 1  # Increment i to avoid infinite loop
}
```

```
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
```

This **while** loop behaves the same as the **for** loop by iterating from 1 to 5 and printing i + 10 in each iteration. However, when `i = 6`, the condition `(i <= 5)` becomes FALSE, and the loop stops. Have a look at Table 2 below to explore the differences between the two types of loop.

Table 2. Key Differences Between `for` and `while` Loops

Feature	**`for` Loop**	**`while` Loop**
Iteration Control	Automatically iterates over a sequence (`1:5`)	Requires manual increment (`i <- i + 1`)
Best Used When	You know the number of iterations	The number of iterations depends on a condition
Risk of Infinite Loop?	No (auto-iterates)	Yes, if increment is missing

6. Mastering Your Daily Routine in R: Essential Steps for a Smooth Workflow

Now that you have learned the fundamental steps and syntax of the R environment, this section will introduce key features essential for building an efficient workflow in your daily routine. While some aspects of R may initially seem complex or unfamiliar, they will become clearer with practice, ultimately helping you streamline tasks and boost productivity.

6.1 Setting the working directory

After launching RStudio, before getting started, it is important to set your working directory. To do so, navigate to the menu at the top of the screen and select **Session > Set Working Directory > Choose Working Directory**, then choose your preferred location. From now, you can load and save your data within this directory.

Alternatively, You can set your working directory in R using the console with the `setwd()` function. Replace "your/path/here" with the actual path to your desired directory. For example:

```
setwd("C:/Users/YourName/Documents/YourProject")
```

To verify that the working directory has been set correctly, use:

```
getwd()
```

This will return the current working directory

6.2 Installing and using packages

Up to this point, we have worked exclusively with built-in functions and packages that come with base R. One of R's greatest strengths is its vast access to a wealth of prewritten functions, contributed by a large community of developers, many of whom are experienced data scientists. These user-created packages have been peer-reviewed and designed for a wide range of statistical analyses. With thousands of packages available, there is often no need to build complex scripts from scratch—it's always worth checking if a package already exists for your task.

While anyone can create and share an R package, most are distributed through CRAN (The Comprehensive R Archive Network: https://cran.r-project.org/), which ensures they meet specific standards before being published. Packages are hosted on CRAN, but they can be installed directly from R's command line.

To install external packages in R, you need to know the package name. Simply use the following command in the console, replacing "NameOfPackage" with the actual package name:

```
install.packages("NameOfPackage")
```

This will download and install the package from CRAN, making it available for use in your R session. For example, let's start by installing the "sf" package, which provides a variety of useful functions for reading and manipulating spatial data. To install it, run the following command in the R console:

```
install.packages("sf")
```

Once your package has been installed, you need to load it in your R session with the following command:

```
library(sf)
```

Once you load a package using the `library()` function, all of its functions, datasets, and help files become accessible for your current R session.

IMPORTANT: every time you begin a new R session, you will need to reload the package with `library()` to use it again. There's no need to re-install it every time because the package, once installed, remains available in your R library for future use. You can also explore all the functions and datasets of a given package by running the command `help(package = "package_name")`. Here an example below:

```
help(package ="sf")
```

6.3 Loading and saving data

One of the most common tasks in R environment is to load, export and save files often in the form of R files and spreadsheets such as Excel or CSV (comma-separated values) files. R provides several built-in functions to handle these tasks efficiently.

Let us export a csv files. First, keep in mind that any file will be exported directly to the current working directory unless a different path is specified. Let us create just a simple matrix and then save it:

```r
# Create a sample matrix
my_matrix <- matrix(1:9, nrow = 3, byrow = TRUE)

# Write the matrix to a CSV file
write.csv(my_matrix, "my_data.csv", row.names = FALSE)
```

Once, you exported this file, you should able to find it in your working directory.

To load the previously saved CSV file back into R, we can use the `read.csv()` function. Let's store the imported data in a new object called "mydata":

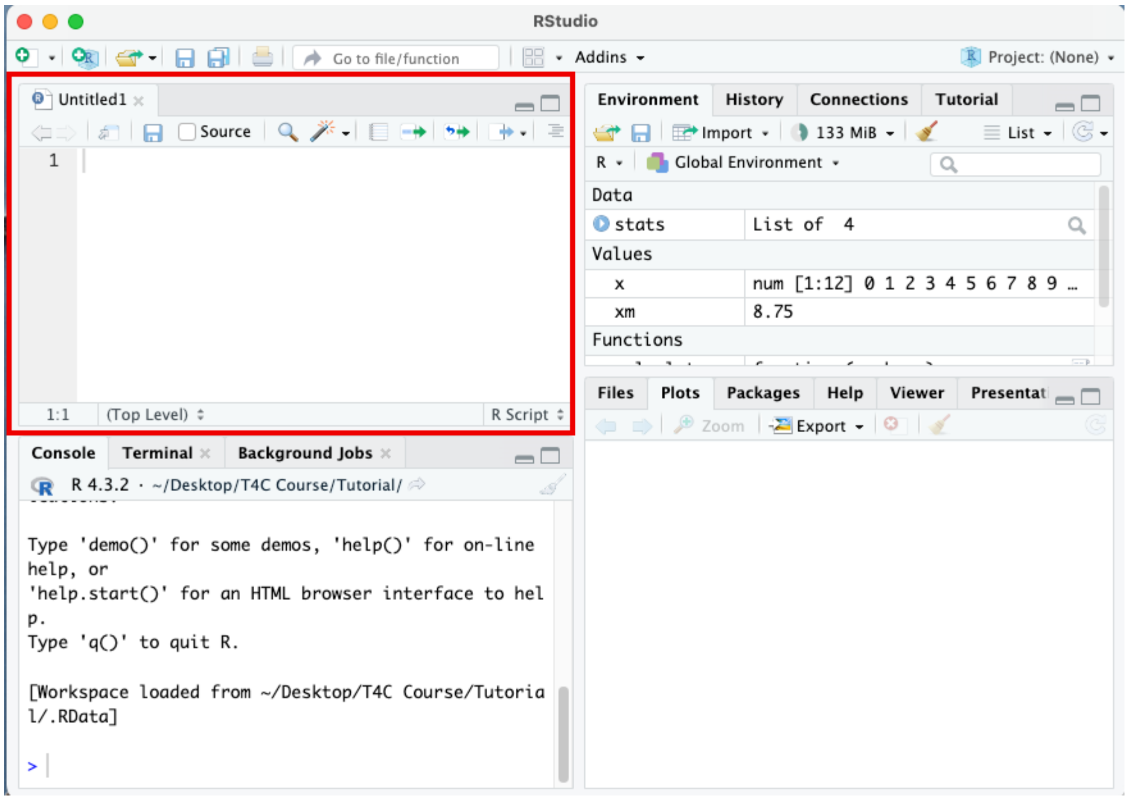
```r
my_data <- read.table("my_data.csv", sep=",", header = TRUE)
```

When using `read.table()` to import data, you can specify the separator with the `sep` argument. This tells R how to distinguish between individual data entries. To find the separator used in your file, you may need to open it in a text editor. If you don't specify the `sep` argument, `read.table()` will assume that white space (such as tabs or spaces) is being used to separate the values, which might not always work correctly, so use this with caution.

Additionally, if your file contains variable names in the first row (instead of actual data values), you should set the `header = TRUE` argument to ensure that R treats the first line as variable names rather than data.

6.4 Writing scripts

You can draft your code while working by utilizing an R script, which is a simple text file for storing R code. In RStudio, you can create a new R script by selecting File > New File > R Script from the menu. This action will open a blank script above the console pane, allowing you to write and organize your code efficiently as shown in Figure 7.



```
Data
stats        List of  4
Values
x            num [1:12] 0 1 2 3 4 5 6 7 8 9 ...
xm           8.75
Functions
```

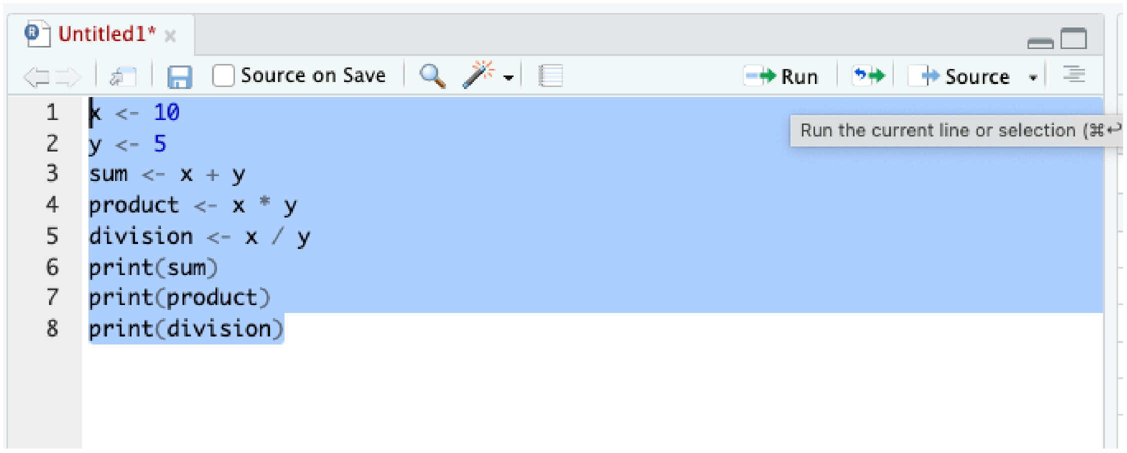
```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/Desktop/T4C Course/Tutorial/.RData]

>
```

It is highly recommended to write and edit your R code in a script before executing it in the console. Doing so ensures you have a reproducible record of your work. By saving your script, you can easily revisit and rerun your analysis later. Additionally, scripts make it easier to review, refine, and share your code with others.

Let us copy the following code chunk below and paste it in the Workspace pane above the console

```
x <- 10
y <- 5
sum <- x + y
product <- x * y
division <- x / y
print(sum)
print(product)
print(division)
```

RStudio offers several built-in tools that simplify working with scripts. One convenient feature is the ability to execute a line of code directly by clicking the “Run” button. The software runs the specific line where the cursor is placed. Additionally, if multiple lines or a section of code are highlighted, RStudio will execute the selected portion as shown in Figure 8 below:



```
x <- 10
y <- 5
sum <- x + y
product <- x * y
division <- x / y
print(sum)
print(product)
print(division)
```

Figure 8. Highlighted portion of code executed by clicking the "Run" button

To save a script, click on the scripts pane, then select "File -> Save As" from the menu bar and put it in your working directory.

6.5 R files

R has the native format .RData, which allows you to store multiple objects. To save data as an RData file, use the save function. The first argument specifies the R object to be saved, followed by a file argument that defines the file name or path. Below an example to store three R objects (a, b, and c) in an RData file:

```
a<-1
b<-c(1:5)
c<-"Hallo World"
save(a,b,c, file = "myobjects.RData")
```

The file will be saved in the working directory you set up. Then, to load the saved file run the command:

```
load("myobjects.RData")
```

7. Summary

You've made significant progress so far. You've started communicating using the R language by writing instructions, executing them via the command line, and customising functions. Occasionally, the computer may provide feedback, such as when an error occurs, but most of the time, it simply performs the requested task and shows the outcome.

The two key elements of R are objects, which store data, and functions, which process and manipulate that data. R also includes several operators (e.g., +, -, *, /, and <-) for performing basic operations. As a data scientist, you'll use R objects to hold data in your computer's memory and R functions to automate processes and perform complex calculations.